US 20040015920A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2004/0015920 A1

Schmidt (43) Pub. Date: Jan. 22, 2004

(54) OBJECT ORIENTED APPARATUS AND METHOD FOR ALLOCATING OBJECTS ON AN INVOCATION STACK IN A DYNAMIC COMPILATION ENVIRONMENT

(75) Inventor: **William Jon Schmidt**, Rochester, MN (US)

Correspondence Address:
MARTIN & ASSOCIATES, LLC
P O BOX 548
CARTHAGE, MO 64836-0548 (US)

(73) Assignee: **International Business Machine Corporation**, Armonk, NY (US)

(21) Appl. No.: 09/812,619

(22) Filed: Mar. 20, 2001

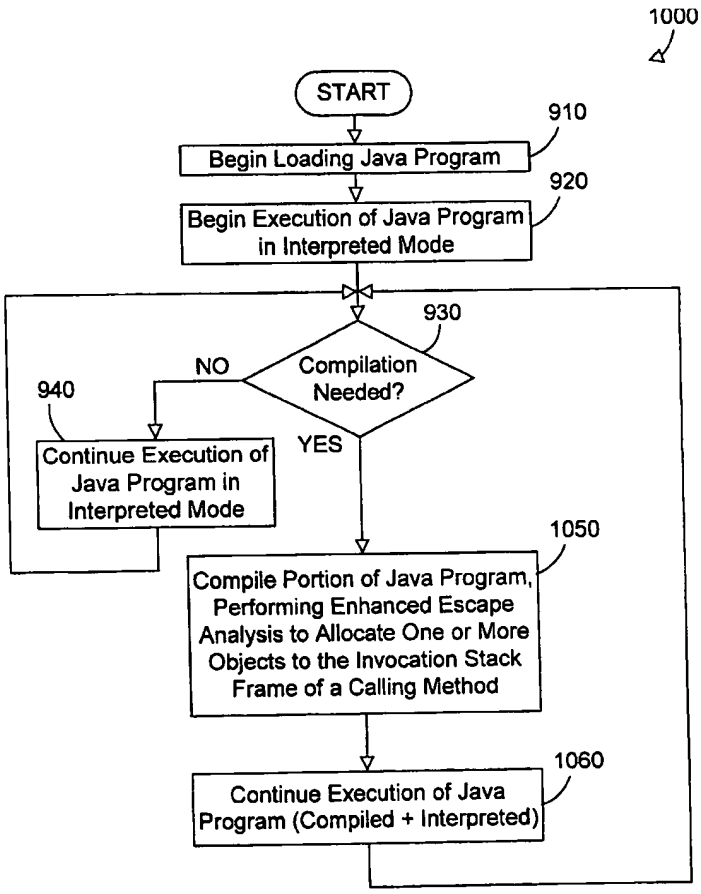
Publication Classification

(51) Int. Cl.[7] G06F 9/45

(52) U.S. Cl. 717/153; 717/162

(57) ABSTRACT

An object oriented mechanism and method allow allocating Java objects on a method's invocation stack in a dynamic compilation environment under certain conditions. When a class is dynamically compiled by a just-in-time (JIT) compiler (as the program runs), one or more of its methods may create objects that may be placed on the method's invocation stack. During the compilation of the class, only the information relating to the previously-loaded classes is taken into account. After compilation, as each new class is loaded, the class is analyzed to see if loading the class might change the analysis used to allocate objects on the invocation stacks of previously-compiled methods. If so, the previous object allocations are analyzed in light of the object reference(s) in the newly loaded class, and the previous object allocations are changed from the invocation stack to the heap, if required. In this manner objects may be allocated to a method's invocation stack based on information that is available from the classes that have been loaded, and can then be changed to be allocated from the heap if information in new classes shows that the previous decision (to allocate on the invocation stack) is no longer valid.



09/812,619
assigned to Chakali
new case

PGPUB-DOCUMENT-NUMBER: 20040015920

PGPUB-FILING-TYPE: new

DOCUMENT-IDENTIFIER: US 20040015920 A1

TITLE: Object oriented apparatus and method for allocating objects on an invocation stack in a dynamic compilation environment

PUBLICATION-DATE: January 22, 2004

INVENTOR-INFORMATION:

NAME	CITY	STATE	COUNTRY
RULE-47			
Schmidt, William Jon	Rochester	MN	US

US-CL-CURRENT: 717/153, 717/162

ABSTRACT:

An object oriented mechanism and method allow allocating Java objects on a method's invocation stack in a dynamic compilation environment under certain conditions. When a class is dynamically compiled by a just-in-time (JIT) compiler (as the program runs), one or more of its methods may create objects that may be placed on the method's invocation stack. During the compilation of the class, only the information relating to the previously-loaded classes is taken into account. After compilation, as each new class is loaded, the class is analyzed to see if loading the class might change the analysis used to allocate objects on the invocation stacks of previously-compiled methods. If so, the previous object allocations are analyzed in light of the object reference(s) in the newly loaded class, and the previous object allocations are changed from the invocation stack to the heap, if required. In this manner objects may be allocated to a method's invocation stack based on information that is available from the classes that have been loaded, and can then be changed to be allocated from the heap if information in new classes shows that the previous decision (to allocate on the invocation stack) is no longer valid.

---------- KWIC ---------

Brief Description of Drawings Paragraph - DRTX (29):
[0042] FIG. 28 is a partial live **call graph** for the sample classes in FIGS. 23-26 that corresponds with the partial class hierarchy graph of FIG. 27;

Brief Description of Drawings Paragraph - DRTX (31):
[0044] FIG. 30 is a partial live **call graph** for the sample classes in FIGS. 23-26 that corresponds with the partial class hierarchy graph of FIG. 29;

Brief Description of Drawings Paragraph - DRTX (39):
[0052] FIG. 38 is a partial live **call graph** that corresponds with the partial class hierarchy graph of FIG. 35;

Detail Description Paragraph - DETX (20):
[0074] A difference between C++ and Java is illustrated in FIGS. 2 and 3. Referring to FIG. 2A, we assume that a "Square" is a name of a particular type of object. A programmer in C++ can declare a variable as being of a "type"

that is an object. As shown in FIG. 2A, the statement "Square k" is a declaration of a variable "k" that is of the type "Square", which is an object. When a variable is defined as an object, as in FIG. 2A, the object can be allocated **directly** on the invocation stack frame for the method, as shown in FIG. 2B. The Square object 230 that corresponds to k is stored on the "invocation stack frame 220 for A.

Detail Description Paragraph - DETX (26):

[0080] The prior art method disclosed in Choi et al. is represented in simplified form in the method 600 of FIG. 6. First, a class hierarchy graph is constructed (step 610). The class hierarchy graph represents inheritance relationships among all classes in a Java program. There is a node in the class hierarchy graph for each class in the program, and there is an arc in the class hierarchy graph from the node for class B to the node for class A if and only if B **directly** inherits from (i.e., "extends") class A.

Detail Description Paragraph - DETX (27):

[0081] Once a class hierarchy graph is constructed in step 610, a live **call graph** is constructed (step 620). The live **call graph** contains one node for every method in a Java program that can apparently be called during that program's execution. Methods that can be called from outside the Java program (such as "main") are designated as "root methods." The node for a method A contains an arc to a subnode for each call site contained in A. There is an arc from the subnode for a call site S to the node for method B if it appears possible for method B to be called at call site S. By definition, every method in a live **call graph** can be reached from at least one root node by a **directed** sequence of arcs; methods that cannot be executed ("dead methods") are not represented in the live **call graph**. A method that calls no other method is designated as a "leaf method." The class hierarchy graph is consulted at **virtual method call** sites to determine which methods may potentially be called, based on inheritance. Construction of class hierarchy graphs and live **call graphs** are well known in the art.

Detail Description Paragraph - DETX (28):

[0082] Once a live **call graph** has been built in step 620, an escape analysis can be performed (step 630). An escape analysis means that each allocation instruction (that creates an object) is labeled as one of the three options: no escape, global escape, and arg escape. Once the escape analysis in step 630 is complete, the code is then generated (step 640) using the information in the escape analysis to determine where each object is allocated. In the prior art Choi et al. approach of method 600, objects that are no escape are allocated on the invocation stack frame of the method that creates the object, while objects that are global escape and arg escape are allocated from the heap.

Detail Description Paragraph - DETX (29):

[0083] Details of the escape analysis step 630 of FIG. 6 are shown in the flow diagram of FIG. 7. First, the methods in the live **call graph** are sorted from the bottom up (step 710). This means that leaf nodes in the graph are considered first, which do not call other methods. The first method on the sorted list is then assigned to M (step 720). A connection graph is then constructed for M, incorporating connection graph information for each method (denoted M.sub.i) that is potentially called from M (step 730). A connection graph denotes potential relationships among variables and parameters that may reference objects, statements that allocate objects, and fields contained in objects. Next, each object allocation in M is marked as global escape, arg escape, or no escape (step 740). If more methods need to be processed (step 750=YES), control is passed to step 720 and processing continues. Once all methods have been processed (step 750=NO), step 630 is done. Note that the description of a connection graph herein is simplified for the purpose of illustrating the preferred embodiments of the invention. For more details

regarding how to construct a connection graph, see the Choi et al. article referenced above.

Detail Description Paragraph - DETX (47):

[0100] A more detailed implementation of a portion of step 1060 of FIG. 10 that is invoked each time a new class is to be loaded is shown in FIG. 14. Another type of memory location that is used by the test-and-set mechanism indicates that a class can be in one of three states: not loaded; in progress; and loaded. Since it would be bad for multiple threads to attempt to load the same class simultaneously, a test-and-set mechanism is used to change the value from "not loaded" to "in progress". The thread that successfully changes the value then finishes the job of loading the class, and finally changes the state from "in progress" to "loaded". Thus, the first step is to mark class C as "in progress" using an atomic test-and-set mechanism (step 1410). If C was previously-marked "in progress" (step 1420=YES), this thread is blocked until C is marked "loaded" (step 1422). In this manner, one thread is given responsibility for loading a class. If C was not previously marked "in progress" (step 1420=NO), C is added to the partial class hierarchy graph (PCHG) (step 1430). If there are no unprocessed methods in C (step 1440=NO), class C is marked as "loaded" (step 1442), and the threads blocked on class C are released (step 1444). If there are unprocessed methods in C (step 1440=YES), the next unprocessed method in C is denoted M (step 1450), and the partial live **call graph** (PLCG) is analyzed to determine whether there are any call sites that could target M (step 1460). If not (step 1460=NO), this method M does not affect the objects that were previously allocated to M's invocation stack, so the next unprocessed method is considered by returning to step 1440. If there is one or more call site in the partial live **call graph** that could target M (step 1460=YES), method M is inserted into the partial live **call graph** and the required analysis is performed to determine the effect of the insertion (step 1470). Control is then passed to step 1440 to determine if there are any more unprocessed methods in C. As soon as all methods have been processed (step 1440=NO), class C is marked as loaded (step 1442), the threads blocked on class C are released (step 1444), and step 1060 is done.

Detail Description Paragraph - DETX (48):

[0101] One suitable implementation of step 1470 in FIG. 14 is shown in FIGS. 15A and 15B, which illustrate what happens when method M is inserted in the partial live **call graph**. First, a node for method M is added in the partial live **call graph** (step 1510). Call sites in M are then added to the partial live **call graph,** with arcs to each possible target in the partial class hierarchy graph, recursively adding nodes for methods that become live as a result (step 1512). Step 1512 creates subnodes for each call site in M, and searches the partial class hierarchy graph to find all loaded methods that could be targets of those call sites. Arcs are added to those methods in the partial live **call graph,** and if any of them become live for the first time, this process is repeated to find call sites in those methods. A set variable denoted "CallerSet" is then created and initialized to be empty (step 1514). The purpose of CallerSet is to build up a set of methods that have previously been analyzed and that contain call sites that may target method M. First, a tuple consisting of the next unprocessed method P and a call site S that could target M is determined (step 1516). An arc is then added from call site S to method M in the partial live **call graph** (step 1520). If P has a previously-constructed connection graph (step 1530=YES), P is added to the CallerSet (step 1532). Otherwise (step 1530=NO), P is not added to the CallerSet. If there remain any unprocessed call sites in the partial live **call graph** that could target method M (step 1540=YES), control is passed to step 1516, and processing continues. If all call sites in the partial live **call graph** that could target M have been processed (step 1540=NO), and if the CallerSet is empty (step 1550=YES), step 1470 is done. However, if the CallerSet is not empty (step 1550=NO), control is passed to step 1140 in FIG. 15B, which is suitably the same as step 1140 in FIGS. 11 and 12, which

constructs a connection graph for method M, using currently loaded classes and methods in the analysis. Next, an escape analysis is performed on M's connection graph (step 630) in the same way as is done in the prior art in a static compilation environment. Appropriate allocation mechanisms for each object allocated in M are then determined (step 740), which is suitably the same as prior art method 740 of FIG. 8. Finally, once the objects have been allocated in step 740, previously-analyzed callers of M must be re-analyzed (step 1590).

Detail Description Paragraph - DETX (56):

[0109] While the invention thus far has been described as computer-implemented methods, the invention could also be practiced as an apparatus that performs the method steps previously discussed. Referring to FIG. 22, a computer system 2200 in accordance with the preferred embodiment is an IBM iSeries 400 computer system. However, those skilled in the art will appreciate that the mechanisms and apparatus of the present invention apply equally to any computer system, regardless of whether the computer system is a complicated multi-user computing apparatus. a single user workstation, or an embedded control system. As shown in FIG. 22, computer system 2200 comprises a processor 2210, a main memory 2220, a mass storage interface 2230, a terminal interface 2240, and a network interface 2250. These system components are interconnected through the use of a system bus 2260. Mass storage interface 2230 is used to connect mass storage devices (such as a **direct** access storage device 2255) to computer system 2200. One specific type of **direct** access storage device 2255 is a floppy disk drive, which may store data to and read data from a floppy disk 2295.

Detail Description Paragraph - DETX (62):

[0115] Terminal interface 2240 is used to **directly** connect one or more terminals 2265 to computer system 2200. These terminals 2265, which may be non-intelligent (i.e., dumb) terminals or fully programmable workstations, are used to allow system administrators and users to communicate with computer system 2200. Note, however, that while terminal interface 2240 is provided to support communication with one or more terminals 2265, computer system 2200 does not necessarily require a terminal 2265, because all needed interaction with users and other processes may occur via network interface 2250.

Detail Description Paragraph - DETX (66):

[0119] Before it can invoke ExampleClass.main( ), the JVM must load ExampleClass (FIG. 26). ExampleClass is marked "in progress" atomically in step 1410 of FIG. 14. A node representing ExampleClass is then added to the partial class hierarchy graph in step 1430. Note that the partial class hierarchy graph starts out containing only those classes loading during JVM startup, with the ExampleClass being the first application class to be loaded. The java.lang.Class class is one of the classes loaded during startup, as shown by the Class node in FIG. 27. Since there is not an explicit extends clause for ExampleClass, its immediate ancestor in the partial class hierarchy graph is java.lang.Object. The methods in ExampleClass are then considered in arbitrary order, according to steps 1440, 1450 and 1460 of FIG. 14. First, we assume that ExampleClass.exampleMethod( ) is checked. Since the partial live **call graph** contains only a call out of the JVM that will target ExampleClass.main( ) (ignoring classes loaded and analyzed during JVM startup), there are no call sites in the partial live **call graph** that can target ExampleClass.exampleMethod( ) **directly**. The same is true of ExampleClass.doSomeWork( ). On the other hand, there is a call site that can target ExampleClass.main( ), so step 1470 causes the steps of FIGS. 15A and 15B to be executed.

Detail Description Paragraph - DETX (67):

[0120] Step 1510 causes ExampleClass.main( ) to be added to the partial live

call graph. Step 1512 adds two call site notes to the partial live **call graph** for the two calls to exampleMethod( ). Since the only target of these call sites is ExampleClass.exampleMethod( ), arcs from the call sites to ExampleClass.exampleMethod( ) are added to the partial live **call graph**. Since this is the first time ExampleClass.exampleMethod( ) has become live, nodes representing the three call sites in ExampleClass.exampleMethod( ) are also added to the partial live **call graph**. The first and third of these have no possible targets in classes loaded so far. For the second, an arc is added to ExampleClass.doSomeWork- ( ). Again, this method has become live for the first time, so nodes for its call sites are added to the partial live **call graph** as well, with arcs to possible targets given the classes loaded so far. FIG. 27 shows the partial class hierarchy graph and FIG. 28 shows the partial live **call graph** to this point.

Detail Description Paragraph - DETX (69):

[0122] Normal JVM activity after loading a class includes a linking phase to combine it into the runtime state of the JVM. Part of this activity is verification, which includes resolution of references. Since ExampleClass of FIG. 26 calls methods of the ComplexNumber class of FIG. 23 and GeneralClass class of FIG. 24, these two classes must also be loaded in a manner similar to what was described above for ExampleClass. Repeating the appropriate steps in FIGS. 14 and 15 results in the partial control hierarchy graph as shown in FIG. 29 and the partial live **call graph** as shown in FIG. 30.

Detail Description Paragraph - DETX (70):

[0123] At this point, execution of ExampleClass.main( ) begins in the JVM's interpreter. The nested loops in this method cause ExampleClass.exampleMethod( ) to be executed 10,000 times. After it has executed 50 times, the JVM determines that exampleMethod( ) should be compiled according to the steps 1050 shown in FIG. 11. Step 1110 marks exampleMethod( ) as "being compiled", assuming that exampleMethod( ) was not previously marked as "being compiled" (step 1120=NO). Since there has not yet been a connection graph constructed for exampleMethod( ) (step 1130=NO), a connection graph for exampleMethod( ) is constructed according to the steps 1140 shown in FIG. 12, which cause the connection graph to be built according to prior art rules that are modified in accordance with the preferred embodiments to ensure that called methods in the partial live **call graph** have connection graphs built first. The primary difference between the escape analysis of the prior art and the escape analysis of the preferred embodiments is that the escape analysis of the preferred embodiments uses partial class hierarchy graphs and partial live **call graphs** to mark allocation instructions as no escape, global escape, or arg escape, whereas the prior art requires a full class hierarchy graph and a full live **call graph** to perform escape analysis. Connection graphs are produced in step 1140 of FIG. 12 for ComplexNumber.<init>( ) and GeneralClass.<init>(which are empty); GeneralClass.examine( ), as shown in FIG. 31; ExampleClass.doSomeWork( ), as shown in FIG. 32; and ExampleClass.exampleMethod( ), as shown in FIG. 33.

Detail Description Paragraph - DETX (71):

[0124] The reader is referred to Choi et al. for a full description of connection graphs, but a few explanatory notes are in order. In FIGS. 31-33, a box represents a parameter or variable that contains object references, and contains the name of that parameter or variable. A circle represents an object allocation instruction. If the circle is dashed, the instruction where the object is allocated is unknown; otherwise the circle is labeled with the instruction. An arc labeled "P" is a points-to arc, meaning the source variable can contain a reference pointer to the objects allocated at the target allocation site. A special node labeled with the "bottom" symbol (.perp.) is used to represent all memory locations outside the current method; if there is a **directed** path from the bottom node to an object allocation instruction, that means that objects allocated at that instruction are global escape.

Detail Description Paragraph - DETX (75):

[0128] Step 1410 marks SpecificClass "in progress". Assuming that SpecificClass was not previously marked "in progress" (step 1420=NO), SpecificClass is then added to the partial class hierarchy graph (step 1430), resulting in the partial class hierarchy graph as shown in FIG. 35. There is an unprocessed method (step 1440=YES), so M is assigned to the next unprocessed method in SpecificClass (step 1450), namely SpecificClass.examine( ). There is a single call site in the partial live **call graph** of FIG. 30 (#5) that could target SpecificClass.examine( ) (step 1460=YES), so the steps in 1470 in FIGS. 15A and 15B are executed. A node for the SpecificClass.examine( ) method is added to the partial live **call graph** (step 1510). SpecificClass.examine( ) does not contain any call sites, so step 1512 has no effect. CallerSet is set to the empty set in step 1514. Step 1516 sets P to ExampleClass.exampleMethod( ) and S to call site #5 in the partial live **call graph** of FIG. 30. An arc from call site #5 to SpecificClass.examine( ) is added to the partial live **call graph** (step 1520), as shown in FIG. 38. Since ExampleClass.exampleMethod( ) has a previously-constructed connection graph (step 1530=YES), ExampleClass.exampleMethod( ) is added to CallerSet (step 1532). This the last relevant call site (step 1540=NO), and CallerSet is not empty (step 1550), so control is passed to step 1140 in FIG. 15B, which builds the connection graph for SpecificClass.examine( ). The resulting connection graph is shown in FIG. 36.